LA-UR -83-3421

CONF-8310182--12

[illegible]

[illegible]

TITLE SAMPLE: SOFTWARE FOR VAX FORTRAN EXECUTION TIMING

AUTHOR(S) Lewis H. Lowe, Analysis and Testing Group (WX-11)

SUBMITTED TO DECUS, Fall 1983 U.S. Symposium Proceedings

MASTER

[illegible]

Los Alamos

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

SAMPLE: SOFTWARE FOR VAX FORTRAN EXECUTION TIMING

Lewis H. Lowe
Analysis and Testing Group (WX-11)
Los Alamos National Laboratory
Los Alamos, New Mexico

ABSTRACT

SAMPLE is a set of subroutines in use at the Los Alamos National Laboratory for collecting CPU timings of various FORTRAN program sections--usually individual subroutines. These measurements have been useful in making programs run faster.

The presentation includes a description of the software and examples of its use.

The software is available on the directory [SAMPLE] of the VAX SIG tape.

INTRODUCTION

When the use of a computer increases, generally the response time increases and programs take longer to run. When this happens, frequently-used programs should be examined to see whether they can be made more efficient.

One approach to code optimization is to look through a listing for code structures that are obviously wasteful. It is better, however, to determine what sections of code require the most CPU time and put effort into optimizing those sections. For example, one can never get a five percent execution speedup by optimizing a section that requires only one percent of the execution time.

A subroutine library called SAMPLE, used to find the CPU time required by a section of code, is described here as well as a program called PROBE, which places SAMPLE routine calls at the entry and exit of each subroutine and function.

The sources for SAMPLE and PROBE, as well as the SAMPLE library, the PROBE executable, and a README file, are available on the directory [SAMPLE] of the VAX SIG tape.

THE SAMPLE LIBRARY

SAMPLE consists of five subroutines: SAMPIN and SAMPDI initialize and terminate the sampling process; SAMPON, SAMPOF, and SAMPSW determine the currently active section of code. Array storage is in a common block SAMPBK, whose length in longwords must be at least five times the number of sections to be timed. The required declaration and calls are shown in Fig. 1 for a program divided into three sections - input, process, and output - with the input section nested in the process section.

Figure 1. Use of the SAMPLE Routines.

```
COMMON /SAMPBK/ IDUMMY(15)
CALL SAMPIN(3)
CALL SAMPON(8HPROCESS )
CALL SAMPON(8HINPUT   )
   (INPUT CODE)
CALL SAMPOF
   (PROCESS CODE)
CALL SAMPSW(8HOUTPUT  )
   (OUTPUT CODE)
CALL SAMPDI(6)
END
```

SAMPIN(MX) must be called to initialize storage. The argument is an upper limit on the number of sections.

SAMPON(NAME) is called to begin timing of a new section. The name of the current section is retained on a stack. The argument is an eight-character field that names the new section.

SAMPOF is called to end timing for the current section, and begin timing the section on the top of the stack; there is no argument.

SAMPSW(NAME) is called to begin timing of a new section without retaining the name of the current section. A call to SAMPSW is like a call to SAMPOF followed by a call to SAMPON but more efficient. The argument is the eight-character name of the new section.

SAMPDI(LU) is called to terminate timing and display the results, as shown in Figs. 2 and 4. The argument is the logical unit number to which the data will be written.

The display contains the name of each section encountered, in alphabetical order. The "samples" column contains the number of ten-millisecond clock ticks apparently charged to the section; the "count" column contains the number of calls to either SAMPON or SAMPSW with the section name as the argument. The "%" column contains an estimate of the percentage of total cpu time used by the section, rounded to the nearest unit. The data used to compute percentages and the estimated CPU time are adjusted in an attempt to subtract off the sampling overhead.

Please note that a section name will not appear in the display unless that section's name appears as the argument of a SAMPON or SAMPSW call.

THE PROBE PROGRAM

A program called PROBE will read a FORTRAN input file and insert SAMPLE calls that set up sections corresponding to subroutines and functions. PROBE will prompt for the input file name (typed without extension) and logical unit number to which the display will be written, separated by a comma.

One pass through the data is made to count the number of programs, functions, subroutines, and entries; at the same time the identifier for each of these blocks is typed. After the list of identifiers, the number of programs, functions, subroutines, and entries is typed.

A second pass through the data is made to write the new file.

In each program, the common block SAMPBK is declared before the first declaration. Also a call to SAMPIN is placed before the first executable statement. If the STOP or CALL EXIT occurs in a logical IF statement, an IF - THEN block is set up.

In each subroutine or function subprogram, a call to SAMPON is placed before the first executable statement and a call to SAMPOF is placed before each RETURN statement. Again, the RETURN statement may appear in a logical IF statement.

Each ENTRY statement is followed by a call to SAMPON and preceded by a call to SAMPOF. In case the ENTRY is unreachable, (for example, immediately following a RETURN statement) the compiler will warn that the CALL SAMPOF statement cannot be reached. This warning may be ignored.

PROBE has been used on several programs written by different programmers, but it cannot handle all FORTRAN IV-PLUS programs. Some possible problems are:

- Unusual blanks, for example, DO 13 I=1,5
- RETURN, STOP, or CALL EXIT in a statement labeled by a DO loop terminator
- Exclamation mark (!) comments
- INCLUDE files containing executable statements.

CASE STUDY 1: 3D MESH CONVERTER

PROBE was used to gather statistics from a utility called SPIN12, which produces a 3D finite element description by spinning a 2D description about the Z axis.

The display is shown in Fig. 2. Most of the time is required by SPIN12, the main program; the only other significant user is PRE02, which converts a 2D element into a ring of 3D elements. Because the main program is straight-line code and PRE02 contains nested loops, the latter was chosen as the optimization candidate.

The subroutine has a loop to read 2D elements and an inner loop to write the number of 3D elements requested by the user. The inner loop contains a test on a fixed value, and some processing on data which changes only in the outer loop. It was a classic and straightforward optimization to take the invariant IF test out of the loops, and move the processing of constant data out of the inner loop. The result of these changes is displayed in Fig. 3. The improvement is measurable but less than satisfactory.

Figure 2. Data Typed by SAMPDI for SPIN12.

Estimated Original Execution Time 74.43.

Interval	Samples	%	Count
FREFIN	4	0	4
IGEiCR	63	1	209
NEWNOD	1887	4	19440
PRE02	2576	35	4
SPIN12	4486	60	1

Figure 3. SPIN12, Before and After.

	Before	After	Improvement
Clock Time	1:13.08	1:07.71	7%
CPU Time	1:09.07	1:06.59	4%
Page Faults	1271	1266	1%

CASE STUDY 2: MODELLING PROGRAM

The next code considered was JENNY, which models continuous systems from their differential equations. Figure 4 shows the display for this program. The two most expensive subprograms, EXPRES and PUTIT, were considered for optimization.

EXPRES is invoked to scan an expression that is part of a 400-character input buffer. There is no end-of-line information in the buffer, and no line length; the buffer is merely blank filled. On finding a blank, the original code tested to column 400; if the end of the line had not been reached, the next column was examined. As a result, many operations were required to detect the end of a line. The modified code, on finding a blank, tests whether the remainder of the line is blank; if so, the line is finished.

PUTIT is a very simple subroutine to enter a character string into a buffer. It contained a form of WHILE loop to find the last nonblank character in the string. This loop was replaced by a FORTRAN 77 DO loop with negative increment; similar changes were made throughout the program.

The results, shown in Fig. 5, are quite pleasing.

Figure 4. Data Typed by SAMPDI for JENNY.

Estimated original execution time 2.31

Interval	Samples	%	Count
DATA	3	1	1
DECLARV	3	1	1
ENAME	2	0	29
EXPRES	77	31	74
FETCH	14	4	59
FINAL	21	9	1
GETLIN	23	9	26
GETSYM	18	0	257
IC	5	2	1
ICDTRM	0	0	1
ICSTRT	0	0	1
JENNY	14	6	1
NAMELU	30	5	231
PREPAR	1	0	24
PROTRM	8	3	22
PUTIT	64	12	454
RIC	1	0	1
SETUP	0	0	1
SOLVEV	6	3	1
SVTERM	9	4	7
SVTMFC	0	0	2
SVTMTM	26	10	28

Figure 5. JENNY: Before and After.

	Before	After	Improvements
Clock Time	0:03.18	0:02.71	15%
CPU Time	0:02.20	0:01.84	16%
Page Faults	200	186	7%

CONSISTENCY

In order to test the consistency of the data, PROBE and SAMPLE were used to measure a simple program competing with interactive programs. The program consists of 1000 calls to each of two subroutines. The first subroutine performs 1000 invocations of AMAX1; the second, 1000 invocations of SQRT.

Figure 6 shows the distribution of estimated CPU times; they all fall within 1 percent of 18.54 seconds.

Figure 7 shows the distribution of the fraction of the time required by the first subroutine. The extreme value falls more than 50 percent above the median of 30 percent.

Good agreement of the former data together with rather poor agreement of the latter suggests that the sampling interval (10 milliseconds) is too long.

Figure 6. Distribution of Estimated CPU Times.

Time	Count
18.39	1
18.41	1
18.47	1
18.50	1
18.54	1
18.55	1
18.58	1
18.64	2
18.69	1

Figure 7. Distribution of Percent Time in Subroutine 1.

Percent	Count
28	1
29	3
30	2
31	1
32	1
34	1
46	1

CONCLUSIONS

SAMPLE and PROBE have proved useful in determining what parts of a program should be considered for optimization.

It is the author's conclusion that the data produced would be more useful if the CPU charge information were available in increments of less than 10 ms.

Finally, traditional optimization methods, such as pulling arithmetic out of loops, appear to be less effective than replacing implicit loops with modern DO loops, which include zero-trip testing and negative increments.